Report

Numerically integrate *cos(x)*

1st March 2021

OVERVIEW

Considering a function f (x) = cos(x) in the interval [− π/2 , π/2]. Writing serial and OpenMP parallel codes to numerically integrate the function using the

(a) Trapezoidal Rule

(b) Montecarlo Method

GOALS

1. To perform a convergence study, using different numbers of divisions (or sampling points), by comparing the integral obtained through the numerical methods with the analytical integral.
2. To perform a timing study using 2, 4, 6 and 8 OpenMP threads and reporting average times of at least 5 runs of the code.

Solving Integration

Analytical integral

Integrating cos(x) in the interval [− π/2 , π/2] as follows,

$$\int_{-\pi/2}^{\pi/2} cos(x)\, dx$$

$$= \; sin(x) \,\Big|_{-\pi/2}^{\pi/2}$$

$$= \; sin\,\frac{\pi}{2} \; - \; sin\,\frac{-\pi}{2}$$

$$= \; 2$$

Trapezoidal Rule

Let we partition $f(x) \in [a, b]$ into n equal subintervals, each of width $\Delta h = \frac{b-a}{n}$

The Trapezoidal Rule for approximating $\int_a^b f(x)\, dx$ is given by,

$$= \frac{\Delta h}{2}\left[f(x_0) + 2f(x_1) + 2f(x_2) + \text{.........} + 2f(x_{n-1}) + f(x_n)\right] \text{......................(1)}$$

where $x_i = a + i\Delta h$ is the sampling point.

Algorithm

- Initialise the number of divisions/samples points for the Trapezoidal Rule.
- Initialise the starting of the current sample as $-\frac{\pi}{2}$.
- Initialise the step size, Δh as $\frac{\pi}{no.\, of\ samples}$.
- Iterate the current sample point as $x_i = -\frac{\pi}{2} + i\Delta h$ for all sample points till $+\frac{\pi}{2}$.
- Compute the result by accumulating the value of $cos(x_i)$ for every sample point such that

  $\sum f \mathrel{+}= cos(x_i)$ according to equation (1).
- Storing the result of integration as the multiplication of step size Δh and the accumulation

  of function$\sum f$.

- To parallelise the code, use **#pragma omp parallel for reduction**(+:$\sum f$) before the

  iteration loop in step 4.

Observations

The result of the trapezoidal rule to numerically integrate and the time taken to solve are reported in the table below.

The time reported is “real time”. The actual real-time spent in running the process from start to finish as if it was measured by a human with a stopwatch.

# of Sample points (n)	Integration Result	Error [Actual - Obtained]	Time Serial Sec	Time 2 threads sec	Time 4 threads sec	Time 6 threads sec	Time 8 threads sec
4	1.896119	0.103881	0.002	0.003	0.003	0.003	0.003
8	1.974232	0.025768	0.002	0.003	0.003	0.003	0.003
12	1.988564	0.011436	0.002	0.003	0.003	0.003	0.003
16	1.993570	0.006430	0.002	0.003	0.003	0.003	0.003
20	1.995886	0.004114	0.002	0.003	0.003	0.003	0.003
50	1.999342	0.000658	0.002	0.003	0.003	0.003	0.003
1e3	1.999998	0.000002	0.002	0.003	0.003	0.003	0.003
1e5	2.000000	0.000000	0.006	0.005	0.004	0.003	0.003
1e8	2.000000	0.000000	1.542	0.811	0.499	0.368	0.291
1e9	2.000000	0.000000	15.211	7.897	4.720	3.362	2.753

Conclusions

- As the number of sampling points increases, the error reduces as shown in Figure1.
- As the number of cores increases, the time to solve reduces as shown in Figure2.

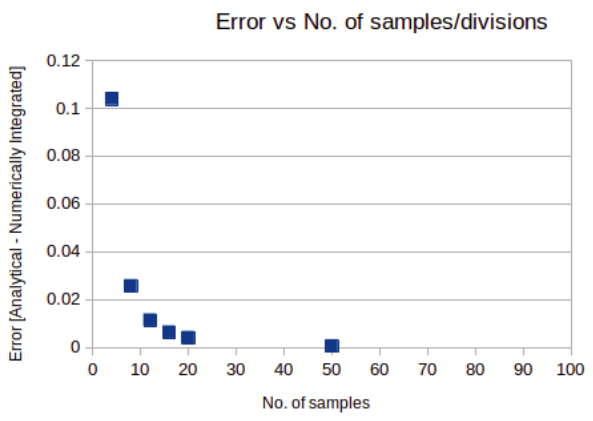

Figure 1

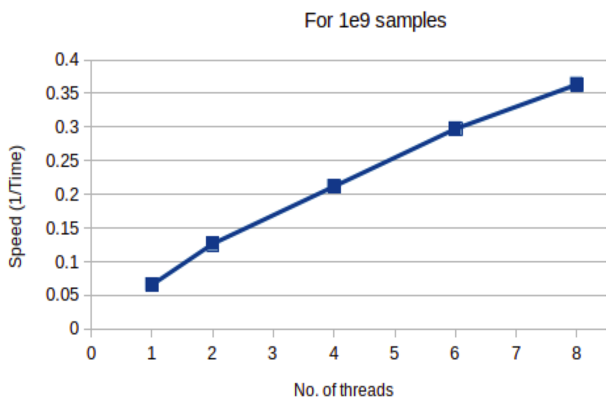

Figure 2

Monte-carlo Method

In the trapezoidal rule of numerical integration, a deterministic approach is used. However, Montecarlo integration employs a non-deterministic approach where each realization provides a different outcome.

Algorithm

- Initialise the number of samples points for the Montecarlo Method.
- Initialise $x_{limit2} = -\frac{\pi}{2}$, $x_{limit1} = \frac{\pi}{2}$ & $y_{limit2} = 1$, $y_{limit1} = 0$.
- Initialise the total area as the multiplication of $Area = [x_{limit1} - x_{limit2}] \times [y_{limit1} - y_{limit2}]$.
- Initialise the seed for random function generation i.e. **rand_r(&seed).**
- Iterate a loop for the number of sample points and using the random number generator function call find the number of x and y-axis samples with-in the range corresponding to $x_{limit2} = -\frac{\pi}{2}$, $x_{limit1} = \frac{\pi}{2}$ & $y_{limit2} = 1$, $y_{limit1} = 0$.
- Compute and accumulate a variable called, **hits** if the cos(x) > y for all samples generated in the above step. Also, compute and accumulate all the iterations as **all_samples_count.**
- Calculate the ratio of hits as $Ratio = \frac{hits}{all\_samples\_count}$, The result of integration will be $Area \times Ratio$.
- To parallelise the code, use **#pragma omp parallel for private (iterator, seed) reduction(+:hits, all_samples_count)** before the iteration loop in step 5.

Observations

The result of the Montecarlo method to numerically integrate and the time taken to solve are reported in the table below.

# of Sample points (n)	Serial Code		Parallel 2 threads		Parallel 4 threads		Parallel 6threads		Parallel 8 threads	
	Result	Time	Result	Time	Result	Time	Result	Time	Result	Time
10	2.513274	0.002	1.570796	0.003	1.884956	0.003	1.570796	0.003	0.628319	0.003
100	1.947788	0.002	2.199115	0.003	1.790708	0.003	1.822124	0.003	2.324779	0.003
1e3	1.988628	0.002	1.944646	0.003	2.016903	0.003	1.982345	0.003	1.935221	0.003
1e5	2.001415	0.011	1.995257	0.007	2.003834	0.005	2.001666	0.004	2.005436	0.004
1e8	2.000115	3.840	1.999941	1.980	1.999780	1.089	1.999750	0.806	1.999780	0.635
1e9	1.999997	38.110	1.999992	19.619	2.000002	10.625	1.999968	7.794	1.999991	6.176

Conclusions

- As the number of sampling points increases, error reduces as shown in the table above.
- As the number of cores increases, the time to solve reduces as shown in Figure 3&4.
- With the thread-safe variant i.e rand_r(), we are making seed as a private variable amongst threads to ensure the state is not shared between threads. It makes the sequences independent of each other. If we start them with private seeds, we will get different sequences in all threads. This is the reason why we are observing different results for the different number of threads.

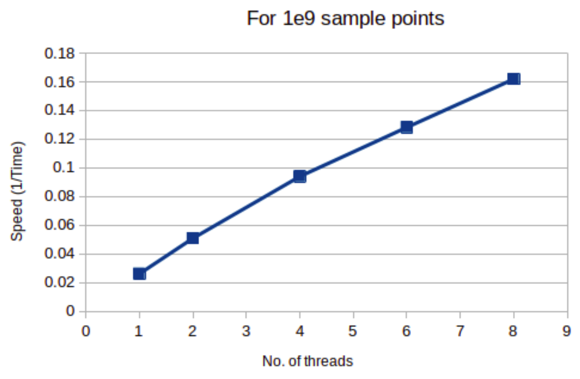

Figure 3

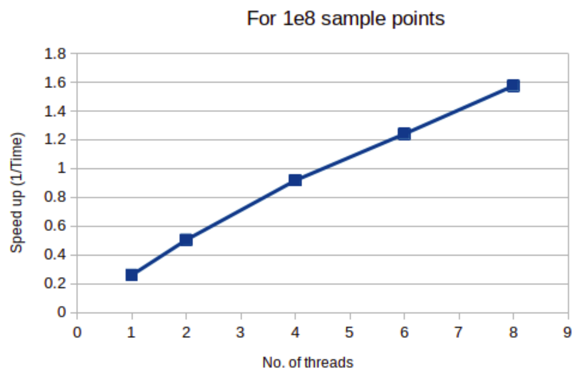

Figure 4